[Prev] [Top] [Next]

2.2 Gate Logic

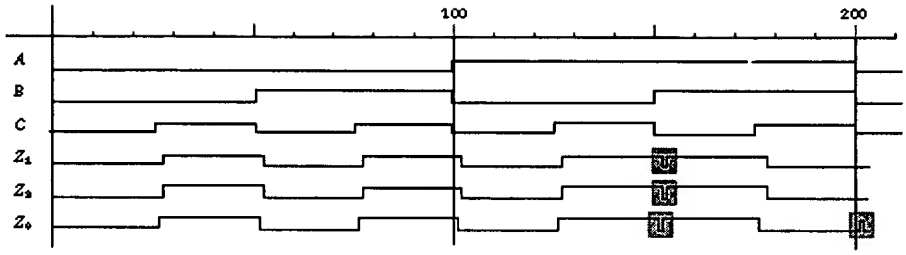


Figure 2.11 Waveform behavior of three implementations of the truth table of Figure 2.9(a).

You now know that there are many gate-level implementations with the same truth table behavior. In this section, you will learn the methods for deriving a reduced gate-level implementation of a Boolean function in two-level form. This usually yields circuits with minimum delay, although gate counts are typically not minimized.

2.2.1 Laws and Theorems of Boolean Algebra

Boolean algebra provides the foundation for all of the simplification techniques we shall discuss. Based on the Boolean laws of Section 2.1, we can prove additional theorems that can be used as tools to simplify Boolean expressions. For example, if $E1$ and $E2$ are two expressions for the same Boolean function, we say that $E2$ is simpler than $E1$ if it contains fewer literals. This usually (but not always) means that the simpler expression will also contain fewer Boolean operations.

Duality Before we provide a tabulation of useful laws and theorems, it is important to describe the concept of duality. Every Boolean expression has a dual. It is derived from the original by replacing AND operations by OR operations and vice versa, and replacing constant **logic** 0's by **logic** 1's and vice versa, while leaving the literals unchanged. It is a fundamental theorem of Boolean algebra, which we do not prove here, that any statement that is true about a Boolean expression is also true for its dual. Once we discover a useful theorem for simplifying a Boolean expression, we obtain its dual as a bonus. For example, the dual of the Boolean theorem $X + 0 = X$, written $(X + 0)$ D, is the theorem $X 1 = X$.

The switch diagrams of Figures 2.2 and 2.3 illustrate the physical basis of the duality between AND and OR operations. The series path of normally open switches in the AND is replaced by parallel paths in the OR. A similar exchange occurs for the paths of normally closed switches. NAND and NOR exhibit the same kind of dual relationships.

Useful Laws and Theorems The following is a list of frequently used laws and theorems of Boolean algebra. Some are generalized from Section 2.1. The second column shows the duals of the expression in the first column.

Operations with 0 and 1:

1. $X + 0 = X$ **1D.** $X\,1 = X$
2. $X + 1 = 1$ **2D.** $X\,0 = 0$

Idempotent theorem:

1. $X + X = X$ **3D.** $X\,X = X$

Involution theorem:

1. $(X')' = X$

Theorem of complementarity:

1. $X + X' = 1$ **5D.** $X\,X' = 0$

Commutative law:

1. $X + Y = Y + X$ **6D.** $X\,Y = Y\,X$

Associative law:

1. $(X + Y) + Z = X + (Y + Z)$ **7D.** $(X\,Y)\,Z = X\,(Y\,Z) = X + Y + Z = X\,Y\,Z$

Distributive law:

1. $X\,(Y + Z) = X\,Y + X\,Z$ **8D.** $X + (Y\,Z) = (X + Y)\,(X + Z)$

Simplification theorems:

1. $X\,Y + X\,Y' = X$ **9D.** $(X + Y)\,(X + Y') = X$
2. $X + X\,Y = X$ **10D.** $X\,(X + Y) = X$
3. $(X + Y')\,Y = X\,Y$ **11D.** $(X\,Y') + Y = X + Y$

DeMorgan's theorem:

1. $(X + Y + Z + \ldots)' = X'\,Y'\,Z'$ **12D.** $(X\,Y\,Z\,\ldots)' = X' + Y' + Z' +$
2. $\{(X1,X2,\ldots,Xn,0,1,+,')\} = \{(X1',X2',\ldots,Xn',1,0,,+)\}$

Duality:

1. $(X + Y + Z + \ldots)\,D = X\,Y\,Z$ **14D.** $(X\,Y\,Z\,\ldots)\,D = X + Y + Z +$
2. $\{(X1,X2,\ldots,Xn,0,1,+,D)\} = (X1,X2,\ldots,Xn,1,0,,+)$

Theorem for multiplying and factoring:

1. $(X + Y)\,X' + Z)$ **16D.** $X\,Y + X'\,Z = (X + Z)\,X' + Y)$

$= X\ Z + X'\ Y$

Consensus theorem:

1. $X\ Y + Y\ Z + X'\ Z = X\ Y + X'\ Z$ **17D.** $(X + Y)\ \ Y + Z)\ \ X' + Z) = (X + Y)\ \ X' + Z)$

The notation $(X1,X2,...,Xn,0,1,+,$ used in theorems 13 and 15 represents an expression in terms of the variables $X1, X2, , Xn$, the constants 0, 1, and the Boolean operations + and . Theorem 13 states succinctly that, in forming the complement of an expression, the variables are replaced by their complements-that is, 0 is replaced by 1 and 1 by 0, and + is replaced by and by +.

Since any of the listed theorems can be derived from the original laws shown in Section 2.1, there is no reason to memorize all of them. The first eight theorems are the most heavily used in simplifying Boolean expressions.

Verifying the Boolean Theorems Each of the theorems can be derived from the axioms of Boolean algebra. We can prove the first simplification theorem, sometimes called the uniting theorem, as follows:

```
X  Y + X  Y'    =      X?

X (Y + Y')      =      X       distributive law (8)

X (1)   =       X       complementarity theorem (5)

X       =       X ÷     identity (1D)
```

As another example, let's look at the second simplification theorem:

```
X + X  Y        =       X?

X  1 + X  Y     =       X       identity (1D)

X (1 + Y)       =       X       distributive law (8)

X (1)   =       X       identity (2)

X       =       X ÷     identity (1)
```

DeMorgan's Theorem DeMorgan's theorem gives a procedure for complementing a complex function. The complemented expression is formed from the original by replacing all literals by their complements; all 1's become 0's and vice versa, and ANDs become ORs and vice versa. This theorem indicates an interesting relationship between NOR, OR, NAND, and AND$\overline{X+Y}= \overline{X}\cdot\overline{Y}\quad \overline{X\cdot Y}= \overline{X}+\overline{Y}$:

Note that $\overline{X+Y}\neq \overline{X}+\overline{Y}$ and $\overline{X\cdot Y}\neq \overline{X}\cdot\overline{Y}$. In other words, NOR is the same as AND with complemented inputs while NAND is equivalent to OR with complemented inputs! This is easily seen to be true from the truth tables of Figure 2.12.

X	Y	$\overline{X}$	$\overline{Y}$	$\overline{X+Y}$	$\overline{X}\cdot\overline{Y}$
0	0	1	1	1	1
0	1	1	0	0	0
1	0	0	1	0	0
1	1	0	0	0	0

X	Y	$\overline{X}$	$\overline{Y}$	$\overline{X}\cdot\overline{Y}$	$\overline{X}+\overline{Y}$
0	0	1	1	1	1
0	1	1	0	1	1
1	0	0	1	1	1
1	1	0	0	0	0

Figure 2.12 DeMorgan's law

Let's use DeMorgan's theorem to find the complement of the following expression: $F = \overline{A}\overline{B}C + \overline{A}B\overline{C} + A\overline{B}\overline{C} + ABC$:

Step by step, the complement is formed as follows

$$\begin{aligned}\overline{F} &= \overline{\overline{A}\overline{B}C + \overline{A}B\overline{C} + A\overline{B}\overline{C} + ABC} \\ &= \overline{\overline{A}\overline{B}C} \cdot \overline{\overline{A}B\overline{C}} \cdot \overline{A\overline{B}\overline{C}} \cdot \overline{ABC} \\ &= (A + B + \overline{C})(A + \overline{B} + C)(\overline{A} + B + C)(\overline{A} + \overline{B} + \overline{C})\end{aligned}$$

Note that duality and DeMorgan's law are not the same thing. The procedure for producing the dual is similar, but the literals are not complemented during the process. Thus, the dual of NOR is NAND (and vice versa); the dual of OR is AND (and vice versa). Remember, any theorem that is true for an expression is also true for its dual.

`Example The Full Adder Carry-Out` We can use the laws and theorems just introduced to verify the simplified expression for the full adder's carry-out function. The original expression, derived from the truth table, is

$$Cout = \overline{A}B\,Cin + A\overline{B}\,Cin + AB\overline{Cin} + ABCin$$

The first step uses theorem 3, the idempotent theorem, to introduce a copy of the term *AB Cin*. Then we use the commutative law to rearrange the terms:

$$\begin{aligned}&= \overline{A}B\,Cin + A\overline{B}\,Cin + AB\overline{Cin} + AB\,Cin + ABCin \\ &= \overline{A}B\,Cin + AB\,Cin + A\overline{B}\,Cin + AB\overline{Cin} + ABCin\end{aligned}$$

We next use the distributive law to factor out the common literals from the first two terms:

$$= (\overline{A} + A)B\,Cin + A\overline{B}\,Cin + AB\overline{Cin} + ABCin$$

We apply the complementarity law:

$$= (1)B\,Cin + A\overline{B}\,Cin + AB\overline{Cin} + ABCin$$

and the identity law:

$$= B\,Cin + A\overline{B}\,Cin + AB\overline{Cin} + ABCin$$

We can repeat the process for the second and third terms. The steps are: (1) idempotent theorem to introduce a redundant term, (2) commutative law to rearrange terms, (3) distributive law to factor out common literals, (4) complementarity theorem to replace $(\overline{B} + B)$ with 1, and (5) identity law to replace 1 X by X:

$$\begin{aligned}&= B\,Cin + A\overline{B}\,Cin + AB\overline{Cin} + ABCin + ABCin \\ &= B\,Cin + A\overline{B}\,Cin + ABCin + AB\overline{Cin} + ABCin \\ &= B\,Cin + A(\overline{B} + B)Cin + AB\overline{Cin} + ABCin \\ &= B\,Cin + A(1)Cin + AB\overline{Cin} + ABCin\end{aligned}$$

The final simplification, using the distributive theorem, complementarity theorem, and identity law, proceeds similarly:

$$\begin{aligned}&= B\,Cin + A\,Cin + AB(\overline{Cin} + Cin) \\ &= B\,Cin + A\,Cin + AB(1) \\ &= B\,Cin + A\,Cin + AB\end{aligned}$$

This is exactly the reduced form of the expression we used in Chapter 1. Although it leads to a simpler expression, applying the rules of Boolean algebra in this fashion does not guarantee you will obtain the simplest expression. A more systematic approach will be introduced in Section 2.3.

Boolean Algebra and Switches We have already observed the close correlation between Boolean functions and switching circuits. Each of the laws and theorems we have discussed has a switching circuit analog. These provide a good way to remember the various theorems.

Some of the switch circuit equivalents are shown in Figure 2.13.

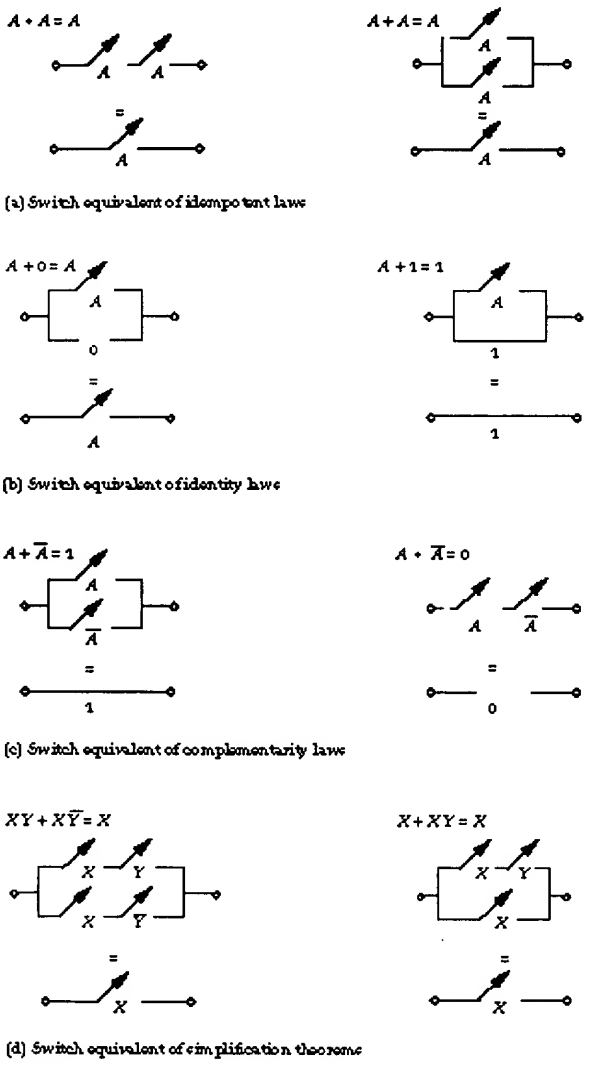


Figure 2.13 Switch equivalent of some laws and theorems of Boolean algebra.

To simplify the diagrams, we show only the paths from true to the output. The idempotent theorems are depicted in Figure 2.13 (a). Obviously, two identically controlled switches, either in series or in parallel, behave as a single switch, since both are open or closed at the same time.

Figure 2.13 (b) shows some of the identity laws. An open connection represents **logic** 0; a shorted connection represents **logic** 1. An open connection in parallel with a normally open switch has no effect on the switching function and can be eliminated. A shorted connection in parallel with a normally open switch guarantees that a connected path always exists to the output. This leaves the function completely independent of the normally open switch.

The complementarity theorems are shown in Figure 2.13 (c). Two switches in parallel, controlled by complementary signals, ensure that one is always open while the other is closed, thus guaranteeing a path to the output. Two switches in series, controlled by complementary signals, guarantee that the connection path is always broken by one of them. This is equivalent to an open connection.

Figure 2.13 (d) shows the switch equivalents of some simplification theorems. In the first network, the

existence of a connection between input and output depends only on X, since Y and $\overline{Y}$ are in parallel and one of the switches will be closed while the other is open. If X is true, the connection is made; if X is false, the connection is broken. In the second network, Y's effect is made redundant by the two X switches in parallel.

2.2.2 Two-Level Logic Canonical Forms

To compare Boolean functions when expressed in algebraic terms, it is useful to have a standard form with which to represent the function. This standard term is called a canonical form, and it is a unique algebraic signature of the function. You will frequently encounter two alternative forms: sum of products and product of sums. We introduce these next.

Sum of Products You have already met the sum of products form in Section 1.3.3. It is also sometimes known as a disjunctive normal form or min-term expansion. A sum of products expression is formed as follows. Each row of the truth table in which the function takes on the value 1 contributes an ANDed term, using the asserted variable if there is a 1 in its column for that row or its complement if there is a 0. These are called **min-terms**. Technically, a min-term is defined as an ANDed product of literals in which each variable appears exactly once in either true or complemented form, but not both. The **min-terms** are then ORed to form the expression for the function. The min-term expansion is unique because it is deterministically derived from the truth table.

A	B	C	F	$\overline{F}$
0	0	0	0	1
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	1	0
1	0	1	1	0
1	1	0	1	0
1	1	1	1	0

Figure 2.14 Sample truth table.

Figure 2.14 shows a truth table for a function and its complement. The min-term expansions for F and $\overline{F}$ are

$$F = \overline{A}BC + A\overline{B}\overline{C} + A\overline{B}C + AB\overline{C} + ABC$$
$$\overline{F} = \overline{A}\overline{B}\overline{C} + \overline{A}B\overline{C} + \overline{A}\overline{B}C$$

We can write such expressions in a shorthand notation using the binary number system to encode the **min-terms**. Figure 2.15 shows the relationship between the truth table row and the numbering of the min-term.

A	B	C	Minterms
0	0	0	$\overline{A}\,\overline{B}\,\overline{C} = m_0$
0	0	1	$\overline{A}\,\overline{B}\,C = m_1$
0	1	0	$\overline{A}\,B\,\overline{C} = m_2$
0	1	1	$\overline{A}\,B\,C = m_3$
1	0	0	$A\,\overline{B}\,\overline{C} = m_4$
1	0	1	$A\,\overline{B}\,C = m_5$
1	1	0	$A\,B\,\overline{C} = m_6$
1	1	1	$A\,B\,C = m_7$

Figure 2.15 Shorthand notation for minterms of three variables.

Note that the ordering of the Boolean variables is critical in deriving the min-term -index. In this case, A determines the most significant bit and C is the least significant bit. You can write the shorthand expression for F and $\overline{F}$ as

$$F\ (A,B,C) = S\ m(3,4,5,6,7) = m3 + m4 + m5 + m6 + m7$$

$$\overline{F}\ (A,B,C) = S\ m(0,1,2) = m0 + m1 + m2$$

where mi represents the ith min-term. The indices generalize for functions of more variables. For example, if F is defined over the variables A, B, C, then $m3$ (0112) is the min-term $\overline{A}BC$. But if F is defined over A, B, C, D, then $m3$ (00112) is $\overline{A}\overline{B}CD$.

The min-term expansion is not guaranteed to be the simplest form of the function, in terms of the fewest literals or terms, nor is it likely to be. You can further reduce the expression for F by applying Boolean algebra:

$$\begin{aligned} F(A, B, C) &= A\bar{B}(C + \bar{C}) + AB\bar{C} + A\bar{B}(C + \bar{C}) \\ &= A\bar{B} + \bar{A}B\bar{C} + AB \\ &= A\bar{B} + AB + \bar{A}B\bar{C} \\ &= A(\bar{B} + B) + \bar{A}B\bar{C} \\ &= A(1) + \bar{A}B\bar{C} \\ &= A + \bar{A}B\bar{C} \\ &= A + B\bar{C} \end{aligned}$$

The one step you may find tricky is the last one, which applies rule 11D, $(X \cdot \bar{Y}) + Y = X + Y$, substituting A for Y and BC for X.

A and BC are called product terms: ANDed strings of literals containing a subset of the possible Boolean variables or their complements. For F defined over the variables A, B, and C, $\bar{A}BC$ is a min-term and a product term, but BC is only a product term.

The minimized gate-level implementation of F is shown in Figure 2.16.

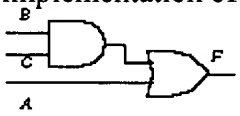


Figure 2.16 Two-level AND-OR gate-level implementation.

Each product term is realized by its own AND gate. The product term A is the degenerate case of a single literal. No AND gate is needed to form this term. The product terms' implementations are then input to a second-level OR gate. The sum of products form leads directly to a two-level realization.

We can repeat the simplification process for $\bar{F}$, but DeMorgan's theorem gives us a good starting point for applying Boolean simplification:

$$\bar{F} = \overline{(A + B\bar{C})} = \bar{A}(\bar{B} + C) = \bar{A}\bar{B} + \bar{A}C$$

Although this procedure is not guaranteed to obtain the simplest form of the complement, it does so in this case.

Product of Sums The involution theorem states that the complement of a Boolean expression's complement is the expression itself. By using DeMorgan's theorem twice, we can derive a second canonical form for Boolean equations. This form is called the product of sums and sometimes the conjunctive normal form or maxterm expansion.

The procedure for deriving a product of sums expression from a truth table is the logical dual of the sum of products case. First, find the rows of the truth table where the function is 0. A maxterm is defined as an ORed sum of literals in which each variable appears exactly once in either true or complemented form, but not both. We form a maxterm by ORing the uncomplemented variable if there is a 0 in its column for that row, or the complemented variable if there is a 1 there. This is exactly opposite to the way we formed **min-terms**. There is one maxterm for each 0 row of the truth table; these are ANDed together at the second level.

The products of sums for the functions F and $\bar{F}$ of Figure 2.14 are $F = (A + B + C)(A + \bar{B} + C)(A + \bar{B} + \bar{C})$ $\bar{F} = (A + B + \bar{C})(\bar{A} + B + C)(\bar{A} + \bar{B} + C)(A + \bar{B} + \bar{C})$

Once again, we often use a shorthand notation.

A	B	C	Maxterms
0	0	0	$A+B+C=M_0$
0	0	1	$A+B+\overline{C}=M_1$
0	1	0	$A+\overline{B}+C=M_2$
0	1	1	$A+\overline{B}+\overline{C}=M_3$
1	0	0	$\overline{A}+B+C=M_4$
1	0	1	$\overline{A}+B+\overline{C}=M_5$
1	1	0	$\overline{A}+\overline{B}+C=M_6$
1	1	1	$\overline{A}+\overline{B}+\overline{C}=M_7$

Figure 2.17 Maxterm shorthand for a function of three variables

Figure 2.17 shows the relationship between maxterms and their shorthand form. We can write F and $\overline{F}$ as

F (A, B, C) = PM (0,1,2) = M0 M1 M2

$\overline{F}$ (A, B, C) = PM (3,4,5,6,7) = M3 M4 M5 M6 M7

where M_i is the ith maxterm.

Interestingly, the maxterm expansion of F could have been formed directly by applying DeMorgan's theorem to the min-term expansion of $\overline{F}$:

$$\overline{F} = \overline{A}BC + A\overline{B}C + AB\overline{C}$$
$$\overline{\overline{F}} = \overline{(\overline{A}BC + A\overline{B}C + AB\overline{C})}$$
$$F = (A + \overline{B} + \overline{C})(\overline{A} + B + \overline{C})(\overline{A} + \overline{B} + C)$$

Of course, the same is true for deriving the min-term form of F from the maxterm form of $\overline{F}$:

$$\overline{F} = (A + \overline{B} + \overline{C})(\overline{A} + B + \overline{C})(\overline{A} + \overline{B} + C)(\overline{A} + \overline{B} + \overline{C})$$
$$\overline{\overline{F}} = \overline{(A + \overline{B} + \overline{C})(\overline{A} + B + \overline{C})(\overline{A} + \overline{B} + C)(\overline{A} + \overline{B} + \overline{C})}$$
$$F = \overline{A}BC + A\overline{B}C + AB\overline{C} + ABC$$

It is easy to translate a product of sums expression into a gate-level realization. The zeroth level forms the complements of the variables if they are needed to realize the function. The first level creates the individual maxterms as outputs of OR gates. The second level is an AND gate that combines the maxterms.

We can find a minimized product of sums form by starting with the minimized sum of products expression of $\overline{F}$. To complement this expression, we use DeMorgan's theorem:

$$\overline{F} = \overline{A}B + \overline{B}C$$
$$\overline{\overline{F}} = \overline{\overline{A}B + \overline{B}C}$$
$$F = (A + \overline{B})(B + C)$$

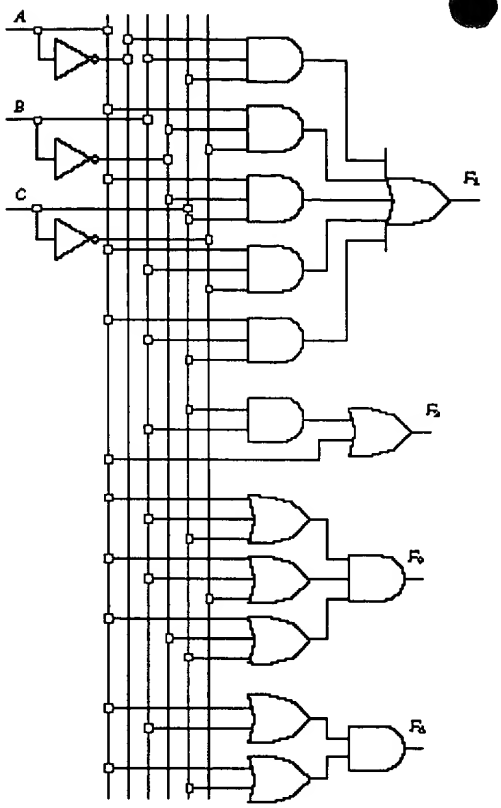


Figure 2.18 Four implementations of F.

Figure 2.18 shows the four different gate-level implementations for F discussed so far: canonical sum of products ($F1$), minimized sum of products ($F2$), canonical product of sums ($F3$), and minimized product of sums ($F4$). In terms of gate counts, the product of sums canonical form is more economical than the sum of products canonical form. But the minimized sum of products form uses fewer gates than the minimized product of sums form. Depending on the function, one or the other of these forms will be better for implementing the function.

To demonstrate that the implementations are equivalent, Figure 2.19 shows the timing waveforms for the circuits' responses to the same inputs. Except for short-duration glitches in the waveforms, their shapes are identical.

Conversion Between Canonical Forms We can place any Boolean function in one of the two canonical forms, sum of products or product of sums. It is easy to map an expression in one canonical form into the other. The procedure, using the shorthand notation we already introduced, is summarized here:

1. To convert from the min-term expansion to the maxterm expansion, you rewrite the min-term shorthand notation to maxterm shorthand, replacing the term numbers with those not used in the min-term list. This is equivalent to applying DeMorgan's theorem to the complement of the function in min-term form. *Example:* $F(A,B,C) = \mathrm{S}\, m(3,4,5,6,7) = \mathrm{P}\, M(0,1,2)$
2. To convert from the maxterm expansion to the min-term expansion, you rewrite the maxterm shorthand notation to min-term shorthand, replacing term numbers with those not used in the maxterm list. This is equivalent to applying DeMorgan's theorem to the complement of the function in maxterm form. *Example:* $F(A,B,C) = \mathrm{P}\, M(0,1,2) = \mathrm{S}\, m(3,4,5,6,7)$

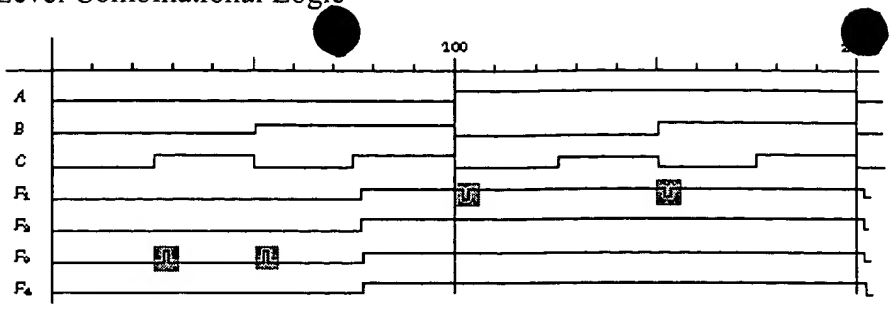


Figure 2.19 Timing waveforms for the four implementations of F.

3. To obtain the min-term expansion of the complement, given the min-term expansion of the function, you simply list the **min-terms** not in *F*. The same procedure works for obtaining the maxterm complement of a function expressed in maxterm form. *Example:*
4. *F* (*A*,*B*,*C*)
   = S *m*(3,4,5,6,7) *F*
   (*A*,*B*,*C*) = P *M*(0,1,2)

   F (*A*,*B*,*C*) =
   S *m*(0,1,2) *F* (*A*,*B*,*C*)
   = P *M*(3,4,5,6,7)

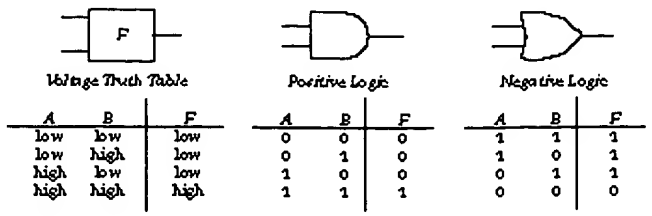
5. To obtain the maxterm expansion of the complement, given the min-term expansion of the function, you simply use the same maxterm numbers as used in *F*'s min-term expansion. The same procedure applies if a min-term expansion of the complement is to be derived from the maxterm expansion of the function. *Example:*
6. *F* (*A*,*B*,*C*)
   = S *m*(3,4,5,6,7) *F*
   (*A*,*B*,*C*) = P *M*(0,1,2)

1. *F* (*A*,*B*,*C*) =
P *M*(3,4,5,6,7) *F* (*A*,*B*,*C*)
= S *m*(0,1,2)

2.2.3 Positive Versus Negative Logic

When you implement **logic** gates as electronic devices, they operate on voltages rather than **logic** levels. There are always two interpretations of any truth table describing the operation of a gate, based on positive or negative **logic**. It is only through a choice of one of these conventions that the voltage levels can be interpreted as **logic** levels. The output voltages are the same; only the logical interpretation is different.

General Concept So far, we have assumed that **logic** 1 is represented by a higher voltage than **logic** 0. This convention is often called active high or positive **logic**. When you want a particular signal to be asserted (for example, "open the garage door"), you place a positive voltage on the -signal and it is interpreted as a **logic** 1. An alternative convention is sometimes more convenient, especially when using NAND and NOR gates to implement **logic** that initiates an event (enable **logic**) or inhibits it from taking place (disable **logic**). It is called active low or negative **logic**. In this case, a low voltage is used to denote that a signal is asserted, while the high voltage is used to represent an unasserted signal.



Voltage Truth Table

A	B	F
low	low	low
low	high	low
high	low	low
high	high	high

Positive Logic

A	B	F
0	0	0
0	1	0
1	0	0
1	1	1

Negative Logic

A	B	F
1	1	1
1	0	1
0	1	1
0	0	0

Figure 2.20 Alternative truth table interpretation in positive and negative logic.

Consider Figure 2.20, which shows a truth table expressed in terms of two relative voltages, high and low. Under the interpretation of positive **logic**, the truth table describes an AND function. But if we interpret the voltage levels according to negative **logic**, we obtain an OR function instead. This follows because an OR function and an AND function are duals, derived by replacing 0's in one truth table with 1's in the other, and vice versa.

Given a function in positive **logic**, we can find its equivalent negative **logic** function simply by applying duality. For example, the dual of the NOR function, $\overline{A+B} = \overline{A} \cdot \overline{B}$, is $\overline{A+B} = \overline{A} \cdot \overline{B}$. Of course, this is the NAND. We can verify this with the truth tables of Figure 2.21.

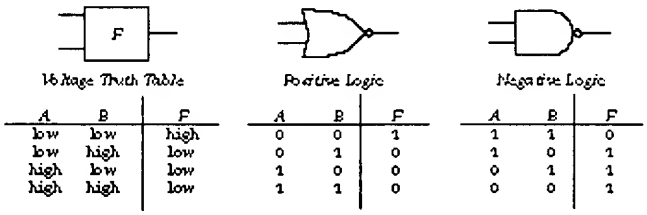


Figure 2.21 Alternative truth table interpretation for N AND and NOR in positive and negative logic.

Example Because of the very real possibilities for confusion, designers prefer to avoid mixing positive and negative **logic** in their designs. However, this is not always possible. For example, a positive **logic** output, asserted high, might connect to a negative **logic** input, asserted low. To illustrate this point, let's take an example from the traffic light controller from Chapter 1.

Your task is to define three signals, Change_Lights, Change_Request, and Timer_Expired, such that Change_Lights is asserted whenever -Change_Request and Timer_Expired are asserted. In positive **logic**/active high notation, the latter two signals should be ANDed to implement Change_Lights. This is shown in Figure 2.22 (a) .

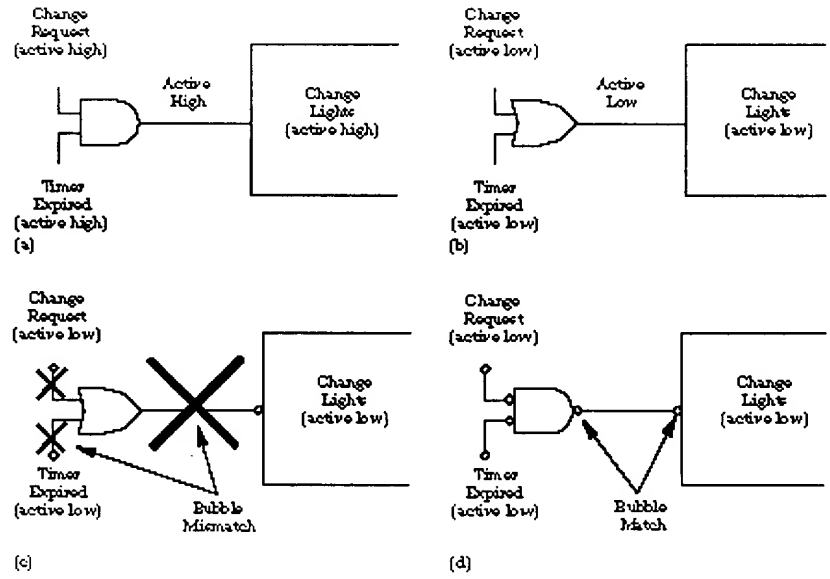


Figure 2.22 Illustration of negative logic and bubble notation.

If you use negative **logic**, and the signals are to be asserted active low, the AND gate is replaced by its dual, an OR gate. Change_Lights is asserted low only when both Change_Request and Timer_Expired are low. In all other cases, Change_Lights is unasserted high. This is shown in Figure 2.22 (b) .

It can be confusing to keep track of whether positive or negative **logic** conventions are being used in a circuit. As an alternative, we adopt a notation that assumes that all gates are positive **logic**, but we explicitly keep track of whether signals are asserted high or asserted low. A "bubble" is placed on the input or output that is to be asserted low.

To make it easier to follow signal polarity within a schematic, active low input bubbles should be matched with active low output bubbles. Continuing with the example, Figure 2.22 (c) shows a case with mismatched bubbles. Starting with Figure 2.22 (d), we add bubbles to indicate the active low polarity of the three signals. You can see that the inputs and the output of the OR gate do not match the polarity of signals to which they are -attached.

How do we make the polarities match? By DeMorgan's theorem, the following is true:

$$A + B = \overline{\overline{A + B}} = \overline{\overline{A} \cdot \overline{B}}$$

An OR gate is equivalent to an AND gate with inverted inputs and outputs. By replacing the OR gate in Figure 2.22 (c) with an AND gate of this kind, we do not change the sense of the **logic** but neatly match up the signal polarities. This is shown in Figure 2.22 (d). The figure clearly indicates that Change_Request and Timer_Expired must both be asserted low to cause the active low Change_Lights signal to become asserted.

2.2.4 Incompletely Specified Functions

We have assumed that we must define an *n*-input function on all of its 2n possible input combinations. This is not always the case. We study the case of incompletely specified functions in this subsection.

`Examples Incompletely Specified Functions` Let's consider a **logic** function that takes as input a binary-coded decimal (BCD) digit. BCD digits are decimal digits, in the range 0 through 9, that are represented by four-bit binary numbers, using the combinations 00002 (0) through 10012 (9). The other combinations, 10102 (10) through 11112 (15), should never be encountered. It is possible to simplify the Boolean expressions for the function if we assume that we do not care about its behavior in these "out of range" cases.

A	B	C	D	W	X	Y	Z
0	0	0	0	0	0	0	1
0	0	0	1	0	0	1	0
0	0	1	0	0	0	1	1
0	0	1	1	0	1	0	0
0	1	0	0	0	1	0	1
0	1	0	1	0	1	1	0
0	1	1	0	0	1	1	1
0	1	1	1	1	0	0	0
1	0	0	0	1	0	0	1
1	0	0	1	0	0	0	0
1	0	1	0	X	X	X	X
1	0	1	1	X	X	X	X
1	1	0	0	X	X	X	X
1	1	0	1	X	X	X	X
1	1	1	0	X	X	X	X
1	1	1	1	X	X	X	X

Figure 2.23 Truth table for BCD increment by 1

Figure 2.23 shows the truth table for a BCD increment by 1 circuit. Each BCD number is represented by four Boolean variables, *A B C D*. The output of the incrementer is represented by four 4-variable Boolean functions, *W X Y Z*.

The output functions have the value "X" for each of the input combinations we should never encounter. When used in a truth table, the value X is often called a don't care. Do not confuse this with the value X reported by many **logic** simulators, where it represents an undefined value or a don't know. Any actual implementation of the circuit will generate some output for the don't-care cases. When used in a truth table, an X simply means that we have a choice of assigning a 0 or 1 to the truth table entry. We should choose the value that will lead to the simplest implementation.

To see that don't cares eventually are replaced by some **logic** value, let's consider the BCD incrementer truth table. The function *Z* looks as if it could be realized quite simply as the function $\overline{D}$. If we choose to implement *Z* in this way, the X's will be replaced by real **logic** values. Since the inputs 10102 through 11112 will never be encountered by the operational circuit, it shouldn't matter which values we assign to those truth table rows.

We choose an assignment that makes the implementation as simple as possible.

Don't Cares and the Terminology of Canonical Forms In terms of the standard S and P notations, **minterms** or maxterms assigned a don't care are written as *di* or *Di*, respectively. Thus the canonical form for *Z* is written as

$$Z = m0 + m2 + m4 + m6 + m8 + d10 + d11 + d12 + d13 + d14 + d15$$

$$Z = M1\ \ M3\ \ M5\ \ M7\ \ M9\ \ D10\ \ D11\ \ D12\ \ D13\ \ D14\ \ D15$$

Proper specification of don't cares is critical for the successful operation of many computer-aided design tools that perform minimization of Boolean expressions. The terminology they use is slightly different from that commonly found in the **logic** design literature, but it is really nothing more than a reformulation of the basic truth table specification.

Let's introduce the concepts by way of the truth table of Figure 2.23. This function is multioutput because it is represented by four output bits defined over the same inputs. It is incompletely specified because it contains don't cares in its outputs. For each of the function's output columns, we can define three sets: the on-set, off-set, and don't-care set. The *on-set* contains all input combinations for which the function is 1. The *off-set* and *don't-care set* are defined analogously for 0 and X, respectively. Thus the on-set for the incrementer's *W* output is { [0,1,1,1], [1,0,0,0] }; its off-set is { [0,0,0,0], [0,0,0,1], [0,0,1,0], [0,0,1,1], [0,1,0,0], [0,1,0,1], [0,1,1,0], [1,0,0,1]; and its don't-care set is { [1,0,1,0], [1,0,1,1], [1,1,0,0], [1,1,0,1], [1,1,1,0], [1,1,1,1] }.

[Prev] [Top] [Next]

This file last updated on 06/23/96 at 19:47:39.
randy@cs.Berkeley.edu;